CoCoNuT: Combining Context-Aware Neural Translation Models using Ensemble for Program Repair

Thibaud Lutellier
tlutelli@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

Hung Viet Pham
hvpham@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

Lawrence Pang
lypang@edu.uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

Yitong Li
yitong.li@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

Moshi Wei
m44wei@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

Lin Tan
lintan@purdue.edu
Purdue University
West Lafayette, IN, USA

ABSTRACT

Automated generate-and-validate (G&V) program repair techniques (APR) typically rely on hard-coded rules, thus only fixing bugs following specific fix patterns. These rules require a significant amount of manual effort to discover and it is hard to adapt these rules to different programming languages.

To address these challenges, we propose a new G&V technique—CoCoNuT, which uses *ensemble* learning on the combination of convolutional neural networks (CNNs) and a new context-aware neural machine translation (NMT) architecture to automatically fix bugs in multiple programming languages. To better represent the context of a bug, we introduce a new context-aware NMT architecture that represents the buggy source code and its surrounding context separately. CoCoNuT uses CNNs instead of recurrent neural networks (RNNs), since CNN layers can be stacked to extract hierarchical features and better model source code at different granularity levels (e.g., statements and functions). In addition, CoCoNuT takes advantage of the randomness in hyperparameter tuning to build multiple models that fix different bugs and combines these models using ensemble learning to fix more bugs.

Our evaluation on six popular benchmarks for four programming languages (Java, C, Python, and JavaScript) shows that CoCoNuT correctly fixes (i.e., the first generated patch is semantically equivalent to the developer's patch) 509 bugs, including 309 bugs that are fixed by none of the 27 techniques with which we compare.

CCS CONCEPTS

• **Computing methodologies → Neural networks**; • **Software and its engineering → Empirical software validation**; **Software defect analysis**; **Software testing and debugging**.

KEYWORDS

Automated program repair, Deep Learning, Neural Machine Translation, AI and Software Engineering

ACM Reference Format:
Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining Context-Aware Neural Translation Models using Ensemble for Program Repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3395363.3397369

1 INTRODUCTION

To improve software reliability and increase engineering productivity, researchers have developed many approaches to fix software bugs automatically. One of the main approaches for automatic program repair is the G&V method [17, 38, 52, 69, 88, 89]. First, candidate patches are generated using a set of transformations or mutations (e.g., deleting a line or adding a clause). Second, these candidates are ranked and validated by compiling and running a given test suite. The G&V tool returns the highest-ranked fix that compiles and passes fault-revealing test cases.

While G&V techniques successfully fixed bugs in different datasets, a recent study [53] showed that very few correct patches are in the search spaces of state-of-the-art techniques, which puts an upper limit on the number of correct patches that a G&V technique can generate. Also, existing techniques require extensive customization to work across programming languages since most fix patterns need to be manually re-implemented to work for a different language.

Therefore, there is a need for a new APR technique that can fix more bugs (i.e., with a better search space) and is easily transferable to different programming languages.

Neural machine translation is a popular deep-learning (DL) approach that uses neural network architectures to generate likely sequences of tokens given an input sequence. NMT uses an *encoder* block (i.e., several layers of neurons) to create an intermediate representation of the input learned from training data and a *decoder* block to decode this representation into the target sequence. NMT has mainly been applied to natural language translation tasks (e.g., translating French to English).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07...$15.00
https://doi.org/10.1145/3395363.3397369

APR can be seen as a translation from buggy to correct source code. Therefore, there is a unique opportunity to apply NMT techniques to learn from the readily available bug fixes in open-source repositories and generate fixes for unseen bugs.

Such NMT-based APR techniques have two key advantages. First, NMT models automatically learn complex relations between input and output sequences that are difficult to capture manually. Similarly, NMT models could also capture complex relations between buggy and clean code that are difficult for manually designed fix patterns to capture. Second, while G&V methods often use hard-coded fix patterns that are programming-language-dependent and require domain knowledge, NMT techniques can be retrained for different programming languages automatically without re-implementing the fix patterns, thus requiring little manual effort.

Despite the great potential, there are two main challenges of applying NMT to APR:

(1) Representing context: How to fix a bug often depends on the context, e.g., statements before and after the buggy lines. However, to represent the context effectively is a challenge for NMT models in both natural language tasks and bug fixing tasks; thus, the immediate context of the sentence to be translated is generally ignored. Two techniques that use NMT to repair bugs [13, 77] concatenate context and buggy code as one input instance. This design choice is problematic. First, concatenating context and buggy code makes the input sequences very long, and existing NMT architectures are known to struggle with long input. As a result, such approaches only fix short methods. For example, Tufano et al. [77] have to focus on short methods that contain fewer than 50 tokens. Second, concatenating buggy and context lines makes it more difficult for NMT to extract meaningful relations between tokens. For example, if the input consists of 1 buggy line and 9 lines of context, the 9 context lines will add noise to the buggy line and prevent the network from learning useful relations between the buggy line and the fix, which makes the models inefficient and less effective on fixing the buggy code.

We propose a new **context-aware NMT architecture** that has two separate encoders: one for the buggy lines, the other one for the context. Using separate encoders presents three advantages. First, the buggy line encoder will only have shorter input sequences. Thus, it will be able to extract strong relations between tokens in the buggy lines and the correct fix without wasting its resources on relations between tokens in the context. Second, a separate context encoder helps the model learn useful relations from the context (such as potential donor code, variables in scope, etc.) without adding noise to the relations learned from the buggy lines. Third, since the two encoders are independent, the context and buggy line encoders can have different complexity (e.g., different number of layers), which could improve the performance of the model. For example, since the context is typically much longer than the buggy lines, the context encoder may need larger convolution kernels to capture long-distance relations, while the buggy line encoder may benefit from a higher number of layers (i.e., a deeper network) to capture more complex relations between buggy and clean lines. *This context-aware NMT architecture is novel and can also be applied to improve other tasks such as fixing grammar mistakes and natural language translation.*

```
- catch (org.mockito.exceptions.verification.junit.
      ArgumentsAreDifferent e) {
+ catch (AssertionError e) {
```

(a) Patch for Mockito 5 in `Defects4J` for Java.

```
  if (actualTypeArgument instanceof WildcardType) {
      contextualActualTypeParameters.put(typeParameter, boundsOf((
            WildcardType) actualTypeArgument));
- } else {
+ } else if (typeParameter != actualTypeArgument) {
      contextualActualTypeParameters.put(typeParameter,
            actualTypeArgument);
  }
```

(b) Patch for Mockito 8 only fixed by a context-aware model.

Figure 1: Two bugs in `Defects4J` fixed by CoCoNuT that other tools did not fix.

(2) Capturing the diversity of bug fixes: Due to the diversity of bugs and fixes (i.e., many different types of bugs and fixes), a single NMT model using the "best" hyperparameters (e.g., number of layers) would struggle to generalize.

Thus, we leverage **ensemble learning** to combine models of different levels of complexity that capture different relations between buggy and clean code. This allows our technique to learn diverse repair strategies to fix different types of bugs.

In this paper, we propose a new G&V technique called **CoCoNuT** that consists of an ensemble of fully convolutional (FConv) models and new context-aware NMT models of different levels of complexity. Each context-aware model captures different information about the repair operations and their context using two separate encoders (one for buggy lines and one for the context). Combining such models allows CoCoNuT to learn diverse repair strategies that are used to fix different types of bugs while overcoming the limitations of existing NMT approaches.

Evaluated against 27 APR techniques on six bug benchmarks in four programming languages, CoCoNuT fixes 509 bugs, 309 of which have not been fixed by any existing APR tools. Figure 1 shows two of such patches for bugs in `Defects4J` [31], demonstrating CoCoNuT's capability of learning *new* and *complex* relations between buggy and clean code from the 3,241,966 instances in the Java training set. CoCoNuT generates the patch in Figure 1a using a **new pattern** (i.e., updating the Error type to be caught) that previous work [12, 27, 29, 35, 37, 38, 45–48, 50, 58, 67–69, 71, 82, 86, 87] did not discover. In the Figure, "-" denotes a line to be deleted that is taken as input by CoCoNuT, while "+" denotes the line generated by CoCoNuT and is identical to the developer's patch. These previous techniques represent **a decade of APR research** that uses manually-designed Java fix patterns. This demonstrates that CoCoNuT complements existing APR approaches by automatically learning new fix patterns that existing techniques did not have, and automatically fixing bugs that existing techniques did not fix. To fix the bug in Figure 1b, CoCoNuT learns from the context lines the correct variables (`typeParameter` and `actualTypeArgument`) to be inserted in the conditional statement.

This paper makes the following contributions:

- A new *context-aware NMT* architecture that represents context and buggy input separately. This architecture is independent of our fully automated tool and could be applied to solve general problems in other domains where long-term context is necessary (e.g., grammatical error correction).

- The first application of *CNN* (i.e., FConv [20] architecture) for APR. We show that the FConv architecture outperforms LSTM and Transformer architectures when trained on the same dataset.
- An *ensemble* approach that combines context-aware and FConv NMT models to better capture the diversity of bug fixes.
- CoCoNuT is the first APR technique that is *easily portable* to different programming languages. With little manual effort, we applied CoCoNuT to four programming languages (Java, C, Python, and JavaScript), thanks to the use of NMT and new tokenization.
- A thorough evaluation of CoCoNuT on six benchmarks in four programming languages. CoCoNuT fixes 509 bugs, 309 of which have not been fixed by existing APR tools.
- A use of attention maps to explain why certain fixes are generated or not by CoCoNuT.

Artifacts are available[1].

2 BACKGROUND AND TERMINOLOGY

Terminology: A DL *network* is a structure (i.e., a graph) that contains nodes or *layers* that are stacked to perform a specific task. Each type of layer represents a specific low-level transformation (e.g., convolution, pooling) of the input data with specific *parameters* (i.e., *weights*). We call DL *architecture* an abstraction of a set of DL networks that have the same types and order of layers but do not specify the number and dimension of layers. A set of *hyperparameters* specifies how one consolidates an architecture to a network (e.g., it defines the convolutional layer dimensions or the number of convolutions in the layer group). The hyperparameters also determine which optimizer is used in training along with the optimization parameters, such as the learning rate and momentum. We call a *model* (or trained model), a network that has been trained, which has fixed weights.

Attention: The attention mechanism [9] is a recent DL improvement. It helps a neural network to focus on the most important features. Traditionally, only the latest hidden states of the encoder are fed to the decoder. If the input sequence is too long, some information regarding the early tokens are lost, even when using LSTM nodes [9]. The attention mechanism overcomes this issue by storing these long-distance dependencies in a separate attention map and feeding them to the decoder at each time step.

Candidate, Plausible, and Correct Patches: We call patches generated by a tool *candidate patches*. Candidate patches that pass the fault triggering test cases are *plausible patches*. Plausible patches that are semantically equivalent to the developers' patches are *correct patches*.

3 APPROACH

Our technique contains three stages: training, inference, and validation. Figure 2 shows an overview of CoCoNuT. In the training phase, we extract tuples of buggy, context, and fixed lines from open-source projects. Then, we preprocess these lines to obtain sequences of tokens, feed the sequences to an NMT network, and tune the network with different sets of hyperparameters. We further train the top-k models until convergence to obtain an ensemble of k models. Since each model has different hyperparameters, each model learns different information that helps fix different bugs.

In the inference phase, a user inputs a buggy line and its context into CoCoNuT. It then tokenizes the input and feeds it to the top-k best models, which each outputs a list of patches. These patches are then ranked and validated by compiling the patched project. CoCoNuT then runs the test suite on the compilable fixes to filter incorrect patches. The final output is a list of *candidate patches* that pass the validation stage.

Section 3.1 presents the challenges of using NMT to automatically fix bugs, while the rest of Section 3 describes the different components of CoCoNuT.

3.1 Challenges

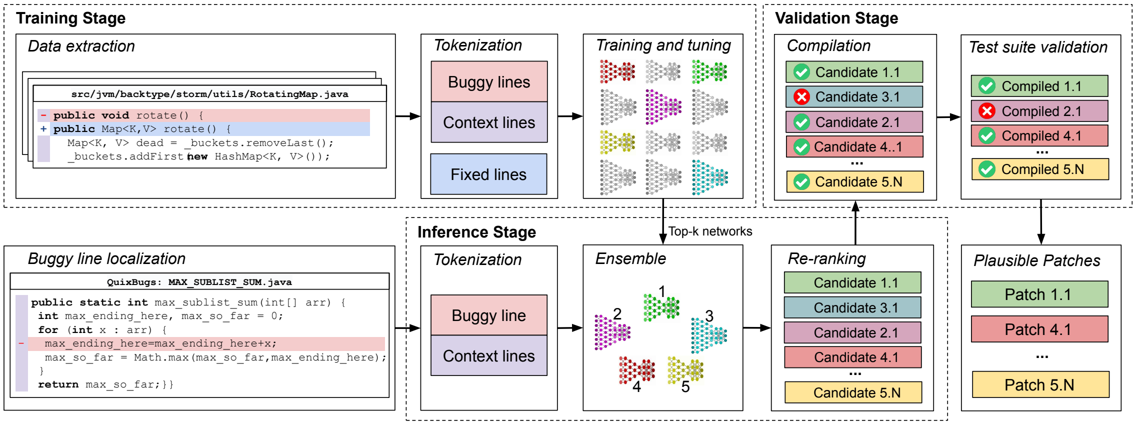
In addition to the two main challenges discussed in Introduction, i.e., **(1) representing context** and **(2) capturing the diversity of fix patterns**, using NMT for APR has additional challenges:

(3) Choice of Layers of Neurons: While natural language text is read sequentially from left to right, source code is generally not executed in the same sequential way (e.g., conditional blocks might be skipped in the execution). Thus, relevant information can be located farther away from the buggy location (e.g., a variable definition can be lines away from its buggy use). As a result, traditional recurrent neural networks (RNN) using LSTM layers [56] may not perform well for APR.

To address these challenges, we build our new context-aware NMT architecture using convolutional layers as the main component of the two encoders and the decoder, as they better capture such different dependencies than RNN layers [16, 20]. We stack convolutional layers with different kernel sizes to represent relations of different levels of granularity. Layers of larger kernel sizes model long-term relations (e.g., relations within a function), while layers of smaller kernel sizes model short-term relations (e.g., relations within a statement). To further track long-term dependencies in both input and context encoders, we use a multi-step attention mechanism.

(4) Large vocabulary size: Compared to traditional natural language processing (NLP) tasks such as translation, the vocabulary size of source code is larger and many tokens are infrequent because developers can practically create arbitrary tokens. In addition, letter case indicates important meanings in source code (e.g., `zone` is a variable and `ZONE` a constant), which increases the vocabulary size further. For example, previous work [13] had to handle a code vocabulary size larger than 560,000 tokens. Practitioners need to cut the vocabulary size significantly to make it scalable for NMT, which leaves a large number of infrequent out-of-vocabulary tokens. We address this challenge by using a new tokenization approach that reduces the vocabulary size significantly without increasing the number of out-of-vocabulary words. For Java, our tokenization reduces the vocabulary size from 1,136,767 to 139,423 tokens while keeping the percentage of tokens out-of-vocabulary in our test sets below 2% (Section 3.3).

[1] https://github.com/lin-tan/CoCoNut-Artifact



```
src/jvm/backtype/storm/utils/RotatingMap.java
- public void rotate() {
+ public Map<K,V> rotate() {
    Map<K, V> dead = _buckets.removeLast();
    _buckets.addFirst(new HashMap<K, V>());
```

```
QuixBugs: MAX_SUBLIST_SUM.java
public static int max_sublist_sum(int[] arr) {
 int max_ending_here, max_so_far = 0;
 for (int x : arr) {
-  max_ending_here=max_ending_here+x;
  max_so_far = Math.max(max_so_far,max_ending_here);
 }
 return max_so_far;}}
```

Figure 2: CoCoNuT's overview

3.2 Data Extraction

We train CoCoNuT on tuples of buggy, context, and fixed lines of code extracted from the commit history of open-source projects. To remove commits that are not related to bug fixes, we follow previous work [81] and only keep commits that have the keywords "fix," "bug," or "patch" in their commit messages. We also filter commits using six commit messages anti-patterns: "rename," "clean up," "refactor," "merge," "misspelling," and "compiler warning." We manually investigate a random sample of 100 commits filtered using this approach and 93 of them are bug-related commits. This is a reasonable amount of noise (7%) for ML training on large training data [34, 85]. We split commits into *hunks* (groups of consecutive differing lines) and consider each hunk as a unique *instance*.

We represent the context of the bug using the function surrounding the buggy lines because it gives semantic information about the buggy functionality, is relatively small, contains most of the relevant variables, and can be extracted for millions of instances. The block "Data Extraction" in Figure 2 shows an example with one buggy line (highlighted in red), one correct line (highlighted in blue), and some lines of context (with purple on the left).

The proposed context-aware NMT architecture is independent of the choice of the context and would still work with different context definitions (e.g., full file or data-flow context). Our contribution is to represent the buggy line and the context separately as a second encoder, independently of the chosen context. Exploring other context definitions is interesting future work.

3.3 Input Representation and Tokenization

CoCoNuT has two separate inputs: a buggy line and its context. The block "Buggy line localization" in Figure 2 shows the inputs for the MAX_SUBLIST_SUM bug in QuixBugs [43]. The line highlighted in red is the buggy line and the lines with a purple block on the left (i.e., the entire function) represent the context. Since typical NMT approaches take a vector of tokens as input, our first challenge is to choose a correct abstraction to transform a buggy line and its context into sequences of tokens.

We use a tokenization method analogous to word-level tokenization (i.e., space-separated), a widely used tokenization in NLP. However, word-level tokenization presents challenges that are specific to programming languages. First, we separate operators from variables as they might not be space-separated. Second, the vocabulary size is extremely large and many words are infrequent or composed of multiple words without separation (e.g., getNumber and get_Number are two different words). To address this issue, we enhance the word-level tokenization by also considering underscores, camel letters, and numbers as separators. Because we need to correctly regenerate source code from the list of tokens generated by the NMT model, we also need to introduce a new token (<CAMEL>) to mark where the camel case split occurs. In addition, we abstract string and number literals except for the most frequent numbers (0 and 1) in our training set.

Thanks to these improvements, we reduce the size of the vocabulary significantly (e.g., from 1,136,767 to 139,423 tokens for Java), while limiting the number of out-of-vocabulary tokens (in our benchmarks, less than 2% of the tokens were out-of-vocabulary).

3.4 Context-Aware NMT Architecture

CoCoNuT's architecture presents two main novelties. The first one consists of using two separate encoders to represent the context and the buggy lines (Figure 3). The second is a new application of fully convolutional layers (FConv) [20] for APR instead of traditional RNN such as LSTM layers used in previous work [13]. We choose the FConv layers because FConv's CNN layers can be stacked to extract hierarchical features for larger contexts [16], which enables modeling source code at different granularity levels (e.g., variable, statement, block, and function). This is closer to how developers read code (e.g., looking at the entire function, a specific block, and then variables). RNN layers model code as a sequence, which is more similar to reading code from left to right.

We evaluate the impact of FConv and LSTM in Section 5.3.

Our architecture consists of several components: an input encoder, a context encoder, a merger, a decoder, and an attention module. For simplicity, Figure 3 only displays a network with one convolutional layer. In practice, depending on the hyperparameters, a complete network has 2 to 10 convolutional layers for each encoder and the decoder.

In training mode, the context-aware model has access to the buggy lines, its context, and the fixed lines. The model is trained to generate the best representation of the transformation from buggy lines with context to fixed lines. In practice, this is conducted by finding the best combination of weights that translates the input instances from the training set to fixed lines. Multiple passes on the training data are necessary to obtain the best set of weights.

In inference mode, since the model does not have access to the fixed line, the decoder processes tokens one by one, starting with a generic <START> token. The outputs of the decoder and the merger are then combined through the multi-step attention module. Finally, new tokens are generated based on the outputs of the attention, the encoders, and the decoder. The generated token is then fed back to the decoder until the <END> token is generated.

Following the example input in Figure 3, a user inputs the buggy statement "`int sum=0;`" and its context to CoCoNuT. After tokenization, the buggy statement (highlighted in red) is fed to the Input encoder while the context (highlighted in purple) is fed to the Context encoder. The outputs of both encoders are then concatenated in the Merger layer.

Since CoCoNuT did not generate any token yet, the token generation starts by feeding the token <START> to the decoder (iteration 0). The output of the decoder (d_{out}) is then combined with the merger output using a dot product to form the first column of the attention map. The colors of the attention map indicate how important each input token is for generating the output. For example, to generate the first token (`double`), the token `int` is the most important input token as it appears in red. The token generation combines the output of the attention, as well as the sum of the merger and decoder outputs ($\Sigma\ e_{out}$ and Σd_{out}) to generate the token `double`. This new token is added to the list of generated tokens and the list is given back as a new input to the decoder (iteration 1). The decoder uses this new input to compute the next d_{out} that is used to build the second column of the attention map and to generate the next token. The token generation continues until <END> is generated.

We describe the different modules of the network below.

Encoders and Decoder: The purpose of the encoders is to provide a fixed-length vectorized representation of the input sequence while the decoder translates such representation to the target sequence (i.e., the patched line). Both modules have a similar structure that consists of three main blocks: an embedding layer, several convolutional layers, and a layer of gated linear units (GLU).

The embedding layers represent input and target tokens as vectors, with tokens occurring in similar contexts having a similar vector representation. In a sense, these layers represent the model's knowledge of the programming language.

The output of the embedding layers is then fed to several convolutional layers. The size of the convolution kernel represents the number of surrounding tokens that are taken into consideration. Stacking such convolutional layers with different kernel sizes provides multiple levels of abstraction for our network to work with. For example, a layer with a small kernel size will only focus on a few surrounding tokens within a statement, while larger layers will focus on a block of code or even a full function. These layers are different in the encoder and the decoder. The encoders use information from both the previous and the next tokens in the input sequence (since the full input sequence is known at all times), while the decoder only uses information about the previously generated tokens (since the next tokens have not been generated yet). Figure 3 shows these differences (full triangles in the encoders and half triangles in the decoder).

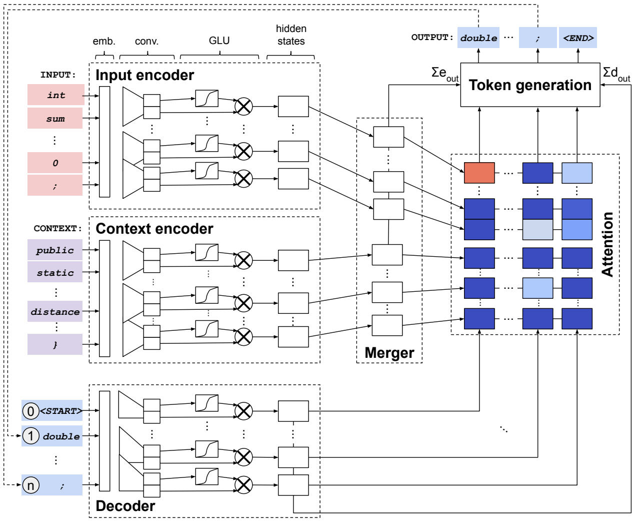


Figure 3: The new NMT architecture used in CoCoNuT.

After the convolutional layers, a layer of GLU (represented by the sigmoid and multiplication boxes in Figure 3) decides which information should be kept by the network. The Merger then concatenates the outputs of the input and context encoders.

Multi-Step Attention: Compared to traditional attention (see Section 2), multi-step attention uses an attention mechanism to connect the output of each convolutional layer in the encoders and the decoder. When multiple convolutional layers are used, it results in multiple attention maps. Multi-step attention is useful because it connects each level of abstraction (i.e., convolutional layer) to the output. This increases the amount of information an encoder passes to the decoder when generating the target tokens.

The attention map represents the impact of input tokens on generating a specific output token and can help explain why a specific output is generated. We analyze attention maps in RQ4.

Token Generation: The token generation combines the outputs of the attention layers, merger ($\Sigma\ e_{out}$) and decoder ($\Sigma\ d_{out}$) to generate the next token. Each token in the vocabulary is ranked by the token generation component based on their likelihood of being the next token in the output sequence. The token selected by the search algorithm is then appended to the list of generated tokens, and the list is sent back as the new input of the decoder. The token generation stops when the <END> token is generated.

Beam Search: We use a common search strategy called beam search to generate and rank a large number of candidate patches. For each iteration, the beam search algorithm checks the t most likely tokens (t corresponds to the beam width) and ranks them by the total likelihood score of the next s prediction steps (s correspond

Table 1: Training set information.

Language	# projects	# instances
Java 2006	45,180	3,241,966
Java 2010	59,237	14,796,149
Python 2010	13,899	480,777
C 2005	12,577	2,735,506
JavaScript 2010	10,163	3,217,093

to the search depth). In the end, the beam search algorithm outputs the top t most likely sequences ordered based on the likelihood of each sequence.

3.5 Ensemble Learning

Fixing bugs is a complex task because there are very diverse bugs with very different fixing patterns that vary in terms of complexity. Some fix patterns are very simple (e.g., changing the operator < to >) while others require more complex modifications (e.g., adding a null checker or calling a different function). Training a model to fix all types of bugs is difficult. Instead, it is more effective to combine multiple specialized models into an ensemble model that will fix more bugs than one single model.

Therefore, we propose an ensemble approach that combines: (1) models with and without context, and (2) models with different hyperparameters (different networks) that perform the best on our validation set.

As described in Section 2, hyperparameters consolidate an architecture to a network. Different hyperparameters have a large impact on the complexity of a network, the speed of the training process, and the final performance of the trained model. For this tuning process, we apply random search because previous work showed that it is an inexpensive method that performs better than other common hyperparameter tuning strategies such as grid search and manual search [10]. For each hyperparameter, based on our hardware limitations, we define a range from which we can pick a random value. Since training a model until convergence is very expensive, we tune with only one epoch (i.e., one pass on the training data). We train n models with different random sets of parameters to obtain models with different behavior and keep the top k best models based on the performance of each model on a separate validation set. This tuning process allows us to discard underfit or overfit models while keeping the k best models that converge to different local optima and fix different real-world bugs.

Finally, we sort patches generated by different models based on their ranks. We use the likelihood of each sequence (i.e., bug fix) generated by each model to sort patches of equal ranks.

3.6 Patch Validation

Statement Reconstruction: A model outputs a list of tokens that forms a fix for the input buggy line. The statement reconstruction module generates a complete patch from the list of tokens. For the abstracted tokens (i.e., strings and numbers), we extract donor code from the original file in which the bug occurred. Once the fix is generated, it is inserted at the buggy location, and we move to the validation step.

Compilation and Test Suite Validation: The model does not have access to the entire project; therefore, it does not know whether the generated patches are compilable or pass the test suite. The validation step filters out patches that do not compile or do not pass the triggering test cases. We use the same two criteria as previous work [89] for the validation process. First, the test cases that make the buggy version pass should still pass on the patched version. Second, at least one test case that failed on the buggy version should pass on the patched version.

3.7 Generalization to Other Languages

Since CoCoNuT learns patterns automatically instead of relying on handcrafted patterns, it can be generalized to other programming languages with minimum effort. The main required change is to obtain new input data. Fortunately, this is easy to do since our training set is extracted from open-source projects. Once the data for the new programming language has been extracted, the top k models can be retrained without re-implementation. CoCoNuT will learn fix patterns automatically for the new programming language.

4 EXPERIMENTAL SETUP

Selecting Training Data: Since the earliest bugs in our Java benchmarks are from 2006 and the latest are from 2016, we divide the instances extracted from the training projects into two training sets. The first one contains all instances committed before 2006 and the second one contains instances committed before 2010. Instances committed after 2010 are discarded. The models trained using the first training set can be used to train models to fix all bugs in the test benchmarks, while the models trained using the second training set should only be used on the bugs fixed after 2010. This setup is to ensure the validity of experiments so that no future data is used [75] (Section 5).

For Java, we use GHTorrent [21] to collect instances from 59,237 projects that contain commits before 2010. We also extract data from the oldest 20,000 Gitlab projects (restriction caused by the limitation of Gitlab search API) because we expect older projects to contain more bugs fixed before the first bug in our benchmarks, and 10,111 Bitbucket repositories (ranked by popularity because of limitations of the Bitbucket search API).

For other languages, we use GHTorrent to extract data from all GitHub projects before the first bug in their associated benchmark. Table 1 shows the number of projects and the number of instances in all training sets.

Selecting State-of-the-art Tools for Comparison: We compare CoCoNuT against 27 APR techniques, including all Java tools used in previous comparisons [47], two additional recent techniques [33, 71], five state-of-the-art C tools [2, 52, 55, 59], and two NMT-based techniques [13, 42]. We chose DLFix [42] and SequenceR [13] over other NMT techniques [23, 60, 77] for comparison, because [77] only generates templates and [23, 60] focus on compilation errors, which is a different problem.

Training, Tuning, and Inference: For tuning, we pick a random sample of 20,000 instances as our validation dataset and use the rest for training.

We use random search to tune hyperparameters. We limit the search space to reasonable values: embedding size (50-500), convolutional layer dimensions (128*(1-5), (1-10)), number of convolutional layers (1-10), dropout, gradient clipping level, and learning rate (0-1). For tuning, we train 100 models for one epoch on the 2006 Java training set with different hyperparameters and rank the hyperparameters sets based on their perplexity [28], which is a standard metric in NLP that measures how well a model generates a sequence. We train the top-k (default k=10) models using ReduceLRonPlateau schedule, with a plateau of 10^{-4}, and stop at convergence or until we reach 20 epochs. In inference mode, we use beam search with a beam width of 1,000.

Infrastructure: We use the Pytorch [65] implementations of LSTM, Transformer, and FConv provided by fairseq-py [1]. We train and evaluate CoCoNuT on three 56-core servers with NVIDIA TITAN V, Xp, and 2080 Ti GPUs.

5 EVALUATION AND RESULTS

Realistic Evaluation Setup: To evaluate CoCoNuT, we use six benchmarks commonly used for APR that contain realistic bugs.

When dealing with time-ordered data, it is not uncommon to incorrectly set up the evaluation [75]. If the historical data used to build the APR technique is more recent than the bugs in the benchmarks, it could contain helpful information (e.g., code clones and regression bugs) that would realistically be unavailable at the time of the fix. Using training/validation instances that are newer than the bugs in the benchmark to train or validate our models would be an incorrect setup and might artificially improve the performances of the models.

This incorrect setup potentially affects all previous APR techniques that use historical data. Although the effect may be smaller, even pattern-based techniques could suffer from this problem since patterns might have been manually learned from bugs that were fixed after the bugs in the benchmarks. To the best of our knowledge, we are the first to acknowledge and address this incorrect setup issue in the context of APR. The using-future-data threat is also one of the reasons that we did not use k-fold cross-validation for evaluation.

To address the setup issue, we extract the timestamp of the oldest bug in the benchmark (based on the date of the fixing commits) and only use training and validation instances before that timestamp. However, adding newer data to the training set would help CoCoNuT fix more recent bugs (e.g., using data until 2010 would be helpful to fix bugs from 2011). A straightforward solution would be to retrain CoCoNuT using different training data for each bug timestamp in the benchmark; however, this is not scalable. Instead, we split our benchmark into two parts. The first part contains bugs from 2006 to 2010 and is used to evaluate CoCoNuT trained with data from before 2006. The second part of the benchmark contains bugs from 2011 to 2016 and is used to evaluate CoCoNuT trained with data from before 2011 (including data from before 2006). This split allows CoCoNuT to learn from instances up to 2010 to fix newer bugs while keeping the overhead reasonable. We then combine the results of CoCoNuT on these two sub-benchmarks to obtain the final number of bugs fixed. With this correct setting, CoCoNuT has no access to data that would be unavailable in a realistic scenario.

Similar to previous work [12, 27, 29, 35, 38, 45–48, 50, 58, 67–69, 71, 82, 86, 87], we stop CoCoNuT after the first generated patch that is successfully validated against the test suite. If no patch passes the test suite after the limit of six hours, we stop and consider the bug not repaired by CoCoNuT. For evaluation purposes only, three co-authors manually compare the plausible patches (i.e., patches that pass the test cases) to the developers' patches and consider a patch *correct* if they all agree it is identical or semantically equivalent to the developers' patch using the equivalence rules described in previous work [49].

5.1 RQ1: How does CoCoNuT perform against state-of-the-art APR techniques?

Approach: Table 2 shows that we compare CoCoNuT with 27 state-of-the-art G&V approaches on six benchmarks for four different programming languages. We use Defects4J [31] and QuixBugs [43] for Java, Codeflaws [76] and ManyBugs [39] for C, and QuixBugs [43] for Python. For JavaScript; we use the 12 examples associated with common bug patterns in JavaScript described in previous work (BugAID) [25]. The total number of bugs in each dataset is under the name of the benchmark.

We compare our results with popular (e.g., GenProg) and recent (e.g., TBar) G&V techniques for C and Java. We do not compare with the JavaScript APR technique Vejovis [64] since it only fixes bugs in Document Object Model (DOM). We extract the results for each technique from a recent evaluation [46] and cross-check against original results when available. Perfect buggy location results are extracted from the GitHub repository of previous work [49] and manually verified to remove duplicate bugs.

We run Angelix, Prophet, SPR, and GenProg on the entire Codeflaws dataset because these techniques had not been evaluated on the full dataset. We use the default timeout values provided by the Codeflaws dataset authors. Since Codeflaws consists of small single-file programs, it is unlikely that these techniques would perform differently with a larger timeout.

The results in Table 2 are displayed as x/y, with x the number of correct patches that are ranked first by an APR technique, and y the number of plausible patches. We also show in parentheses the number of bugs fixed by CoCoNuT that have not been fixed by other techniques. '-' indicates that a technique has not been evaluated on the benchmark or does not support the programming language of the benchmark. For the BugAID and ManyBugs benchmarks, we could not run the validation step; therefore we only display the number of patches that are identical to developers' patches († in the Table) and cannot show the number of plausible patches.

Since different approaches used different fault localization (FL) techniques, we separate them based on FL types (Column FL), as was done in previous work [48]. Standard FL-based approaches use a traditional spectrum-based fault localization technique. Supplemented FL-based APR techniques use additional methods or assumptions to improve FL. For example, HD-Repair assumes that the buggy file and method are known. Finally, Perfect FL-based techniques assume that the perfect localization of the bug is known. According to recent work [46, 49], this is the preferred way to evaluate G&V approaches, as it enables fair assessment of APR techniques independently of the fault localization approach used.

Table 2: Comparison with state-of-the-art G&V approaches. The number of bugs that only CoCoNuT fixes is in parentheses (309). The results are displayed as x/y, with x the number of bugs correctly fixed. and y the number of bugs with plausible patches. * indicates tools whose manual fix patterns are used by TBAR. Numbers are extracted from either the original papers or from previous work [49] which reran some approaches with perfect localization. In Defects4J, we exclude Closure 63 and Closure 93 from the total count since they are duplicates of Closure 62 and 92, respectively. † indicates the number of patches that are identical to developer patches—the minimal number of correct patches. - indicates tools that have not been evaluated on a specific benchmark. The highest number of correct patches for each benchmark is in bold.

		Java		C		Python	JavaScript
FL	Tool	`Defects4J`	`QuixBugs`	`Codeflaws`	`ManyBugs`	`QuixBugs`	`BugAID`
		393 bugs	40 bugs	3,902 bugs	69 bugs	40 bugs	12 bugs
Standard	1 Angelix [59]	-	-	318/591	**18/39**	-	-
	2 Prophet [55]	-	-	301/839	15/39	-	-
	3 SPR [52]	-	-	283/783	11/38	-	-
	4 Astor* [58]	-	6/11	-	-	-	-
	5 LSRepair [45]	19/37	-	-	-	-	-
	6 DLFix [42]	29/65	-	-	-	-	-
Supplemented	7 JAID [12]	9/31	-	-	-	-	-
	8 HD-Repair* [37]	13/23	-	-	-	-	-
	9 SketchFix* [27]	19/26	-	-	-	-	-
	10 ssFix* [86]	20/60	-	-	-	-	-
	11 CapGen* [82]	21/25	-	-	-	-	-
	12 ConFix [33]	22/92	-	-	-	-	-
	13 Elixir* [69]	26/41	-	-	-	-	-
	14 Hercules [71]	49/72	-	-	-	-	-
Perfect	15 SOSRepair [2]	-	-	-	16/23	-	-
	16 Nopol [88]	2/9	1/4	-	-	-	-
	17 (j)Kali [58, 68]	2/8	1/2	-	3/27	-	-
	18 (j)GenProg [38, 58]	6/16	0/2	[255-369]/1423	2/18	-	-
	19 RSRepair [67]	10/24	2/4	-	2/10	-	-
	20 ARJA [91]	12/36	-	-	-	-	-
	21 SequenceR [13]	12/19	-	-	-	-	-
	22 ACS [87]	16/21	-	-	-	-	-
	23 SimFix* [29]	27/50	-	-	-	-	-
	24 kPAR* [46]	29/56	-	-	-	-	-
	25 AVATAR* [48]	29/50	-	-	-	-	-
	26 FixMiner* [35]	34/62	-	-	-	-	-
	27 TBar [47]	**52/85**	-	-	-	-	-
	CoCoNuT (not fixed by others)	44/85 **(6)**	**13/ 20** **(10)**	**423/ 716** **(271)**	7 †/- **(0)**	**19/21** **(19)**	**3** †/- **(3)**
	Total bugs fixed by CoCoNuT			**509** **(309)**			

We exclude iFixR [36] because it uses kPAR to generate fixes which is already in the Table. We choose kPAR since its evaluation is similar to our evaluation setup and allows for a fairer comparison. iFixR proposes to use bug reports instead of test cases. However, we believe that it is reasonable to keep test cases for validation because, even if they were committed at a later stage, they were often available to the developers at the time of the bug report since developers generally discover bugs by seeing a program fail given one failing test case. Regardless, this is still a fair comparison with the 27 existing techniques which all use test cases for validation.

TBar is a recent pattern-based technique that uses patterns implemented by previous work (techniques marked with a * in Table 2). *TBar shows how a combination of most existing pattern-based techniques behaves with perfect localization.*

Results: Overall, Table 2 shows that CoCoNuT fixes 509 bugs across six bug benchmarks in four programming languages. CoCoNuT is the best technique on four of the six benchmarks and is the only technique that fixes bugs in Python and JavaScript, indicating that CoCoNuT is easily portable to different programming languages with little manual effort.

On the **Java** benchmarks, CoCoNuT outperforms existing tools on the `QuixBugs` benchmark, fixing 13 bugs, including 10 bugs that have not been fixed before. On `Defects4J`, CoCoNuT performs better than all techniques but TBar and Hercules. In addition, 6 bugs CoCoNuT fixes have not been fixed by any other techniques. Two of the 6 bugs require new fix patterns that have not been found by any previous work from a decade of APR research. We investigate these six bugs in detail in RQ2. In addition, fifteen of the bugs fixed by Hercules are multi-hunk bugs (i.e., bugs that need changes in multiple locations to be fixed), currently out of scope for CoCoNuT which mostly generates single-hunk bug fixes. In the future, the method used by Hercules to fix multi-hunk bugs could be applied to CoCoNuT.

Two of the 240 `Defects4J` bugs from after 2010 can only be fixed by models trained with the training set containing data up to 2010. Surprisingly, the models trained with data from before 2006 stay relevant and can fix bugs introduced 4 to 10 years after the training data. This highlights the fact that, while training DL

```
-  static float toJavaVersionInt(String version) {
+  static int toJavaVersionInt(String version) {
```

Figure 4: CoCoNuT's Java Patch for Lang 29 in Defects4J that have not been fixed by other techniques.

```
- int indexOfDot=namespace.indexOf('.');
+ int indexOfDot=namespace.lastIndexOf('.');
```

(a) Java patch for Closure 92 in Defects4J.

```
- int end = message.indexOf(templateEnd,start);
+ int end = message.lastIndexOf(templateEnd,start);
```

(b) Similar change to Closure 92 bug occurring in the training data.

Figure 5: Example of patches fixed only by CoCoNuT and related changes in the training set.

```
- while True:
+ while queue:
```

(a) Python patch for BREADTH_FIRST_SEARCH in QuixBugs.

```
- while (true) {
+ while(!queue.isEmpty()) {
```

(b) Java patch for BREADTH_FIRST_SEARCH in QuixBugs.

Figure 6: BREADTH_FIRST_SEARCH bugs fixed by CoCoNuT in both Python and Java QuixBugs benchmarks.

models is expensive, such models stay relevant for a long time and do not need expensive retraining.

In Defects4J, 43% (19 out of 44) of the bugs fixed by CoCoNuT are not fixed by TBar (the best technique), hence CoCoNuT complements TBar very well. There are also several bugs fixed by TBar that CoCoNuT fails to fix. Seven of them are if statement insertions (e.g., null check), and three of them are fixes that move a statement to a different location. These bugs are difficult to fix for CoCoNuT because we targeted bugs that are fixed by modifying a statement, hence these two transformations do not appear in our training set. Fixing bugs by inserting if statements has been widely studied [47, 52, 87, 88]. Instead, we propose a new technique that fixes bugs that are more challenging for other techniques to fix.

While we generate 1,000 patches for each bug per model, correct patches are generally ranked very high. Ten of the 44 correct patches in Defects4J are ranked first by one of the models. The average rank of a correct patch is 63 and the median rank is 4. The worst rank of a correct patch is 728.

A potential threat to validity is that some tools use different fault localization techniques. While we cannot re-implement all existing tools using perfect localization (e.g., some tools are not publicly available), we try our best to mitigate this threat. First, we consider 13 tools that also use perfect localization for comparison, including TBar, which fixed the most number of bugs in Defects4J. Second, TBar uses fix patterns from 10 tools (marked with * in Table 2) with perfect localization. Thus, although some of these 10 tools do not use perfect fault localization, we indirectly compare with these tools using perfect fault localization. CoCoNuT fixes 6 bugs that have not been fixed by existing tools including TBar.

On the **C** benchmarks, CoCoNuT fixes 430 bugs, 271 of which have not been fixed before. CoCoNuT outperforms existing techniques on the Codeflaws dataset, fixing 105 more bugs than Angelix and has a 59% precision (423 out of 716). On the ManyBugs benchmark, CoCoNuT fixes seven bugs, outperforming GenProg, Kali, and RSRepair but is outperformed by other techniques.

We manually check for semantic equivalence for all generated patches except for GenProg on ManyBugs. Instead, we manually check a random sample of 310 out of the 1,423 plausible patches for Codeflaws generated by GenProg, because GenProg rewrites the program before patching it, e.g., replacing a for loop with a while loop, which makes it difficult to manually investigate all 1,423 plausible patches. The margin of error for this sample is 4% with a 95% confidence level, thus, Table 2 shows the projected range of the number of Codeflaws bugs that GenProg fixes.

On the **JavaScript** and **Python** bug benchmarks, CoCoNuT fixes three and 19 bugs respectively.

Since different bug benchmarks contain different distributions of different types of bugs, and different APR tools fix different types of bugs, it is important to evaluate new APR techniques on different benchmarks for a fair and comprehensive comparison. CoCoNuT is the best technique on four of the six benchmarks.

Summary: CoCoNuT is the first approach that has been successfully applied without major re-implementation to different programming languages, fixing **509 bugs** in six benchmarks for four popular programming languages, **309** of which that have not been fixed by existing work, including six in Defects4J, a dataset that has been heavily used to evaluate 22 other tools.

5.2 RQ2: Which bugs only CoCoNuT can fix?

Approach: For the bugs only CoCoNuT can fix in Defects4J, we rerun TBar, the best technique for Java, on our hardware, with perfect localization, and without a time limit to confirm that TBar cannot fix these bugs, even under the best possible conditions. For C, as stated in RQ1, we run Angelix, SPR, Prophet, and GenProg on Codeflaws with the same hardware we used for CoCoNuT for a fair comparison. CoCoNuT is the only technique fixing bugs in the Python and JavaScript benchmarks.

Results: CoCoNuT fixes bugs by automatically learning new patterns that have not yet been discovered, despite a decade of research on Java fix patterns. Mockito 5 (Figure 1a) is a bug in Defects4J only CoCoNuT fixes because it requires patterns that are not covered by existing pattern-based techniques (i.e., exception type update). Lang 29 (Figure 4) is another bug that cannot be fixed by other tools because existing pattern-based techniques such as TBar do not have a pattern for updating the return type of a function. TBar cannot generate any candidate patches for these two bugs.

Thanks to the context-aware architecture, CoCoNuT fixes bugs other techniques do not fix by correctly extracting donor code (e.g., correct variable names) from the context, while techniques such as TBar rely on heuristics. An example of such bugs is in Figure 1b as explained in Section 1.

CoCoNuT is the only technique that fixes Closure 92 because it learns from historical data (Figure 5b) that the lastIndexOf method in the String library is a likely candidate to replace the indexOf method. Other techniques such as TBar fail to generate a correct patch because the donor code is not in their search space.

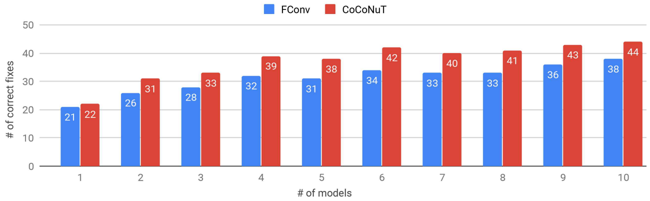


Figure 7: Number of bugs fixed as the number of models considered in ensemble learning increases.

```
- Calendar c = new GregorianCalendar(mTimeZone);
+ Calendar c = new GregorianCalendar(mTimeZone , mLocale);
```

Figure 8: CoCoNuT's Java patch for Lang 26 in `Defects4J`.

CoCoNuT directly learns patterns from historical data without any additional manual work, making it portable to different programming languages. Figure 6 shows the **same** bug in both the Java (Figure 6a) and Python (Figure 6b) implementations of the `QuixBugs` dataset. CoCoNuT fixes both bugs, even if the patches in Java and Python are quite different. Adapting pattern-based techniques for Java to Python would require much work because the fix-patterns are very different, even for the same bug.

Summary: CoCoNuT fixes **309** bugs that have not been fixed by existing techniques, including six bugs in `Defects4J`. CoCoNuT fixes new bugs by (1) automatically learning **new patterns** that have not been found by previous work, (2) extracting donor code **from the context of the bug**, and (3) extracting donor code **from the historical training data**.

5.3 RQ3: What are the contributions of the different components of CoCoNuT and how does it compare to other NMT-based APR techniques?

Approach: To understand the impact of each component of CoCoNuT, we investigate them individually. More specifically, we focus on three key contributions: (1) the performance of our new NMT architecture compared to state-of-the-art NMT architectures, (2) the impact of context, and (3) the impact of ensemble learning. We compare CoCoNuT with DLFix [42] and SequenceR [13], two state-of-the-art NMT-based APR techniques and three other state-of-the-art NMT architectures (i.e., LSTM [56], Transformer [78], and FConv [20]). These models have not been used for program repair, so we implemented them in the same framework as our work (i.e., using Pytorch [65] and the fairseq-py [1] library). To ensure a fair comparison, we tune and train the LSTM, Transformer, and FConv similarly to CoCoNuT. SequenceR [13] uses an LSTM encoder-decoder approach to repair bugs automatically. We use the numbers reported by SequenceR and DLFix's authors on the `Defects4J` dataset for comparison since working versions of SequenceR and DLFix were unavailable at the time of writing. DLFix [42] uses a new treeRNN architecture that represents source code as a tree.

Comparison with State-of-the-art NMT: CoCoNuT fixes the most number of bugs, with 44 bugs fixed in `Defects4J`. DLFix is the second best, fixing 29 bugs, followed by FConv with 21 bugs while Transformer and SequenceR have similar performances, fixing 13 and 12 bugs respectively. The last baseline, LSTM, performs poorly with only 5 bugs fixed. These results demonstrate that CoCoNuT performs better than state-of-the-art DL approaches and that directly applying out-of-the-box deep-learning techniques for APR is ineffective.

Impact of the Context: Figure 7 shows the total number of bugs fixed using the top-k models, with k from 1 to 10. For all k, CoCoNuT outperforms an ensemble of FConv models trained without using context. Our default CoCoNuT (with k = 10 models) fixes six more bugs than using models without context (44 versus 38).

Advantage of Ensemble Learning: Figure 7 shows that as k increases from 1 to 10, the number of bugs that CoCoNuT fixes increases from 22 to 44, a 50% improvement. We observe a similar trend for the FConv models, indicating that ensemble learning is beneficial independently of the architecture used. Model 9 and Model 7 are the best FConv and Context-aware models respectively, both fixing 26 bugs.

While increasing k also increases the runtime of the technique, this cost is not prohibitive because CoCoNuT is an offline technique and can be run overnight. In the worse case, with k=10, CoCoNuT takes an average of 16.7 minutes to generate 20,000 patches for one bug (for k=1, it takes an average of 1.8 minutes per bug).

While CoCoNuT fixes 44 in `Defects4J`, the average number of bug fixed by a single model is 15.65. in `Defects4J`, 40% of the bugs are fixed by five or fewer models. Six of the correctly fixed bugs are only fixed by one model, while only two bugs are fixed by all models. This indicates that different models in our ensemble approach specialize in fixing different bugs.

Summary: The new NMT architecture we propose performs significantly better than baseline architectures. In addition, using ensemble learning to combine the models improves the results, with a 50% improvement for k=10 compared to k=1.

5.4 RQ4: Can we explain why CoCoNuT can (or fail to) generate specific fixes?

The Majority of the Fixes are not Clones: By learning from historical data, the depth of our neural network allows CoCoNuT to fix complex bugs, including the ones that require generating new variables. As discussed at the start of Section 5, we only keep training and validation instances from before the first bugs in our benchmarks for a fair evaluation. As a result, the exact same bug cannot appear in our training/validation sets and evaluation benchmarks. However, the same patch may still be used to fix different bugs introduced at different times in different locations. Having such patch clones in both training and test sets is valid, as recurring fixes are common. The majority of the bugs fixed by CoCoNuT do not appear in the training sets: only two patches from the C benchmark and one from the JavaScript benchmark appear in the training or validation sets. This indicates that CoCoNuT is effective in learning and generating completely different fixes.

Analyzing the Attention Map: CoCoNuT can also fix bugs that require complex changes. For example, the fix for *Lang 26* from

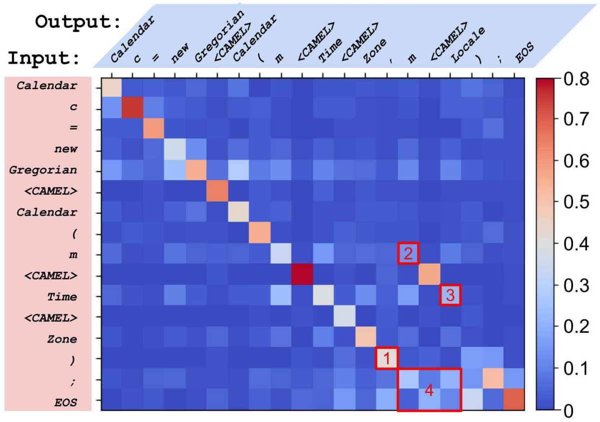


Figure 9: Attention map for the correct patch of *Lang 26* from the `Defects4J` benchmark generated by CoCoNuT.

the `Defects4J` dataset shown in Figure 8 requires injecting a new variable `mLocale`. This new variable only appears four times in our training set, and never in a similar context. However, `mLocale` contains the token `Locale` which co-occurs in our training set with the buggy statement tokens `Gregorian`, `Time`, and `Zone`.

The attention map in Figure 9 confirms that the token `Time` is important for generating the `Locale` variable. Specifically, the tokenized input is shown on the y-axis while the tokenized generated output is displayed on the x-axis. The token `<CAMEL>` between `m` and `Locale` indicates that these two tokens form one unique variable. The attention map shows the relationship between the input tokens (vertical axis) and the generated tokens (horizontal axis). The color in a cell represents the relationship of corresponding input and output tokens. The color scale, shown on the right of the figure, varies from 0 (dark blue) to 0.6 (dark red) and indicates the contribution of each input token for generating an output token.

This attention map helps us understand why the model generates specific tokens. For example, the second `m` in the output is generated because of the token `m` in the input (part of `mTimeZone`), showing that the network can keep the naming convention when generating new variables (square labeled 2 in Figure 9). The token `Locale` is mostly generated because of the token `Time`, indicating that the network is confident these tokens should be together (square 3). Finally, the tokens forming the variable `mLocale` are all influenced by the input tokens `)` `;` and `EOS`, indicating that this token is often used right before the end of a statement (i.e., as the last parameter of the function call, rectangle 4 on Figure 9). This example shows how the attention map can be used to understand why CoCoNuT generates a specific patch.

Limitations of Test Suites: Patches that pass the test suite but are incorrect are an issue that CoCoNuT and other APR techniques share. CoCoNuT could generate a correct fix for 8 additional bugs if more time is given; however, for these 8 bugs, CoCoNuT generates an incorrect patch that passes the test suite (we only validate the first candidate patch due to time considerations) or timed out before the correct fix. Using enhanced test suites proposed in previous work [89] may alleviate this issue.

5.5 Execution Time

Data Extraction: Extracting data from open-source repositories and training CoCoNuT are one-time costs that can be amortized across many bugs and should not be counted in the end-to-end time to fix one bug. For Java, extracting data from 59,237 projects takes five days using three servers.

Training Time: The median time to train our context-aware NMT model for 1 epoch during tuning is 8.7 hours. On average, training a model for 20 epochs takes 175 hours on one GPU. Transformers and FConv networks are faster to train, taking an average of 2.5 and 2.7 hours per epoch. However, training the LSTM network is much slower (22 hours per epoch).

This one time cost is to be compared to **the decade of research** spent designing and implementing new fix patterns.

Cost to fix one bug: Once the model is trained, the main cost is the inference (i.e., generating patches) and the validation (i.e., running the test suite). During inference, generating 20,000 patches for one bug (CoCoNuT default setup) takes 16.7 min on average using one GPU. On our hardware, CoCoNuT's median execution time to validate a bug is six sec on `Codeflaws` and 6 min on `Defects4J` (benchmark with the largest programs and test suites).

CoCoNuT median **end-to-end time** to fix a bug varies from 16.7 min on `Codeflaws` to 22.7 min on `Defects4J`. In comparison, on identical hardware, the median time of other tools that we ran on `Codeflaws` (Angelix, GenProg, SPR, and Prophet) varies from 30 sec to 4 min. TBar, on the same hardware, has an 8 min median execution time on `Defects4J`.

While the end-to-end approach of CoCoNuT is slower than existing approaches, it is still reasonable. In addition, we can shorten the execution time by reducing the number of patches generated in inference. Generating only 1,000 patches (i.e., 50 patches per model) would reduce CoCoNuT's end-to-end time to 2 min on `Codeflaws` and 7 min on `Defects4J` while still fixing most of the bugs (e.g., 34 in `Defects4J`). Parallelism on multiple GPUs and CPUs can also speed up the inference.

6 LIMITATIONS

The dataset used to train our approach is different from the ones used by other work (e.g., SequenceR and DLFix), which could impact our comparison results. However, the choice of datasets and how to extract and represent data are a key component of a technique. In addition, we compare our approach with LSTM, FConv, and Transformer architectures using the same training data and hardware, which shows that CoCoNuT outperforms all other three. Finally, both DLFix and SequenceR use data that was unavailable at the time of the bug fix (i.e., their models are trained using instances committed after the bugs in Defects4J were fixed), which could produce false good results as shown by prior work [75].

A challenge of deep learning is to explain the output of a neural network. Fortunately, for developers, the repaired program that compiles and passes test cases should be self-explanatory. For users who build and improve CoCoNuT models, we leverage the recent multi-step attention mechanism [20] to explain why a fix was generated or not.

There is a threat that our approach might not be generalizable to fixing bugs outside of the tested benchmarks. We use six different benchmarks in four different programming languages to address this issue. In the future, it is possible to evaluate our approach in additional benchmarks [24, 57, 70].

There is randomness in the training process of deep-learning models. We perform multiple runs and find that the randomness in training has little impact on the performances of the trained models.

7 RELATED WORK

Deep Learning for APR: SequenceR [13], DLFix [42], and Tufano et al. [77] are the closest work related to CoCoNuT. The main differences with CoCoNuT are that these approaches use RNN (a single LSTM-based NMT model for SequenceR and Tufano et al., and a TreeRNN architecture for DLFix) and represent both the buggy line and its context as one input. These approaches have trouble extracting long term relations between tokens and do not capture the diversity of bug fixes. We showed in Section 5.3 that CoCoNuT outperforms both DLFix and SequenceR. Tufano et al. [77] generates templates instead of complete patches and thus cannot be directly compared. Deep learning has also been used to detect and repair small syntax [73] and compilation [23, 60] issues (e.g., missing parenthesis). These models show promising results for fixing compilation issues but only learn the syntax of the programming language.

G&V Program Repair: Many APR techniques have been proposed [8, 12, 27, 29, 37, 38, 50, 52, 54, 58, 64, 69, 82, 86–88]. We use a different approach compared to these techniques, and as shown in Section 5, our approach fixes bugs that existing techniques have not fixed. In addition, these techniques require significant domain knowledge and manually crafted rules that are language-dependent, while thanks to our context-aware ensemble NMT approach, CoCoNuT automatically learns such patterns and is generalizable to several programming languages with minimal effort.

Grammatical Error Correction (GEC): Recent work uses machine translation to fix grammar errors [14, 15, 18, 19, 30, 32, 51, 63, 72, 74, 90]. Among them, [15] applied an attention-based convolutional encoder-decoder model to correct sentence-level grammatical errors. CoCoNuT is a new application of NMT models on source code and programming languages, addressing unique challenges. Studying whether our new context-aware NMT architecture improves GEC remains future work.

Deep Learning in Software Engineering: The software engineering community had applied deep learning to perform various tasks such as defects prediction [40, 79, 81], source code representation [6, 7, 66, 80], source code summarization [4, 22] source code modeling [5, 11, 26, 83], code clone detection [41, 84], and program synthesis [3, 44, 61, 62]. Our work uses a new deep learning approach for APR.

8 CONCLUSION

We propose CoCoNuT, a new end-to-end approach using NMT and ensemble learning to automatically repair bugs in multiple languages. We evaluate CoCoNuT on six benchmarks in four different programming languages and find that CoCoNuT can repair 509 bugs including 309 that have not been fixed before by existing techniques. In the future, we plan to improve our approach to work on multi-hunk bugs.

ACKNOWLEDGMENT

The authors thank Shruti Dembla for her contribution in collecting java projects from GitHub. The research is partially supported by Natural Sciences and Engineering Research Council of Canada, a Facebook research award, and an NVIDIA GPU grant.

REFERENCES

[1] Fairseq-py. https://github.com/pytorch/fairseq, 2018.

[2] Afsoon Afzal, Manish Motwani, Kathryn Stolee, Yuriy Brun, and Claire Le Goues. Sosrepair: Expressive semantic search for real-world program repair. *IEEE Transactions on Software Engineering*, 2019.

[3] Carol V Alexandru. Guided code synthesis using deep neural networks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1068–1070. ACM, 2016.

[4] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pages 2091–2100, 2016.

[5] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.

[6] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=BJOFETxR-.

[7] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40, 2019.

[8] Moumita Asad, Kishan Kumar Ganguly, and Kazi Sakib. Impact analysis of syntactic and semantic similarities on patch prioritization in automated program repair. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 328–332. IEEE, 2019.

[9] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[10] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[11] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. Tree2tree neural translation model for learning source code changes. *arXiv preprint arXiv:1810.00314*, 2018.

[12] Liushan Chen, Yu Pei, and Carlo A Furia. Contract-based program repair without the contracts. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*, pages 637–647. IEEE, 2017.

[13] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2019.

[14] Shamil Chollampatt and Hwee Tou Ng. A Multilayer Convolutional Encoder-Decoder Neural Network for Grammatical Error Correction. 2018. URL http://arxiv.org/abs/1801.08831.

[15] Shamil Chollampatt and Hwee Tou Ng. A multilayer convolutional encoder-decoder neural network for grammatical error correction. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, February 2018.

[16] Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 933–941. JMLR.org, 2017.

[17] Thomas Durieux and Martin Monperrus. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test*, pages 85–91. ACM, 2016.

[18] Tao Ge, Furu Wei, and Ming Zhou. Reaching Human-level Performance in Automatic Grammatical Error Correction: An Empirical Study. (3):1–15, 2018. URL http://arxiv.org/abs/1807.01270.

[19] Tao Ge, Furu Wei, and Ming Zhou. Fluency Boost Learning and Inference for Neural Grammatical Error Correction. *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, 1:1–11, 2018. URL http://aclweb.org/anthology/P18-1097.

[20] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. pages 1243–1252, 2017.

[21] Georgios Gousios and Diomidis Spinellis. Ghtorrent: Github's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 12–21, 2012.

[22] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, pages 933–944. ACM, 2018.

[23] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *AAAI*, pages 1345–1351, 2017.

[24] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Bugsjs: A benchmark of javascript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101. IEEE, 2019.

[25] Quinn Hanam, Fernando S de M Brito, and Ali Mesbah. Discovering bug patterns in javascript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 144–156. ACM, 2016.

[26] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE*, pages 763–773, 2017.

[27] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Sketchfix: a tool for automated program repair approach using lazy candidate generation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 888–891. ACM, 2018.

[28] F. Jelinek, R. L. Mercer, L. R. Bahl, and J. K. Baker. Perplexity – a measure of the difficulty of speech recognition tasks. *Journal of the Acoustical Society of America*, 62:S63, November 1977. Supplement 1.

[29] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. pages 298–309, 2018.

[30] Marcin Junczys-Dowmunt, Roman Grundkiewicz, Shubha Guha, and Kenneth Heafield. Approaching Neural Grammatical Error Correction as a Low-Resource Machine Translation Task. (2016):595–606, 2018. URL http://arxiv.org/abs/1804.05940.

[31] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[32] Masahiro Kaneko, Yuya Sakaizawa, and Mamoru Komachi. Grammatical Error Detection Using Error- and Grammaticality-Specific Word Embeddings. *Proceedings ofthe The 8th International Joint Conference on Natural Language Processing*, (2016):40–48, 2017. URL https://github.com/kanekomasahiro/grammatical-error-.

[33] Jindae Kim and Sunghun Kim. Automatic patch generation with context-based change application. *Empirical Software Engineering*, 24(6):4071–4106, 2019.

[34] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 481–490. IEEE, 2011.

[35] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *arXiv preprint arXiv:1810.01791*, 2018.

[36] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. ifixr: bug report driven program repair. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 314–325, 2019.

[37] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. 1:213–224, 2016.

[38] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2012.

[39] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.

[40] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. Software defect prediction via convolutional neural network. In *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on*, pages 318–328. IEEE, 2017.

[41] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. Cclearner: A deep learning-based clone detection approach. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 249–260. IEEE, 2017.

[42] Yi Li, Wang Shaohua, and Tien N. Nguyen. DLfix: Context-based code transformation learning for automated program repair. In *Software Engineering (ICSE), 2020 IEEE/ACM 42nd International Conference on*. IEEE, 2020.

[43] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 55–56. ACM, 2017.

[44] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*, 2016.

[45] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F Bissyandé. Lsrepair: Live search of fix ingredients for automated program repair. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 658–662. IEEE, 2018.

[46] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 102–113. IEEE, 2019.

[47] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 31–42, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6224-5. doi: 10.1145/3293882.3330577. URL http://doi.acm.org/10.1145/3293882.3330577.

[48] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–12. IEEE, 2019.

[49] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé François D Assise Bissyande, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *42nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 2020.

[50] Xuliang Liu and Hao Zhong. Mining stackoverflow for program repair. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 118–129. IEEE, 2018.

[51] Zhuo Ran Liu and Yang Liu. Exploiting Unlabeled Data for Neural Grammatical Error Detection. *Journal of Computer Science and Technology*, 32(4):758–767, 2017. ISSN 18604749. doi: 10.1007/s11390-017-1757-4.

[52] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178. ACM, 2015.

[53] Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 702–713. IEEE, 2016.

[54] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, volume 2017, 2017.

[55] Fan Long et al. *Automatic patch generation via learning from successful human patches*. PhD thesis, Massachusetts Institute of Technology, 2018.

[56] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[57] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. Bears: An extensible java bug benchmark for automatic program repair studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 468–478. IEEE, 2019.

[58] Matias Martinez and Martin Monperrus. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 441–444. ACM, 2016.

[59] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701. ACM, 2016.

[60] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. Deepdelta: learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 925–936, 2019.

[61] Lili Mou, Rui Men, Ge Li, Lu Zhang, and Zhi Jin. On end-to-end program generation from user intention by deep neural networks. *arXiv preprint arXiv:1510.07211*, 2015.

[62] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=HkfXMz-Ab.

[63] Courtney Napoles and Chris Callison-Burch. Systematically Adapting Machine Translation for Grammatical Error Correction. *Proceedings of the 12th Workshop on Innovative Use of NLP for Building Educational Applications*, pages 345–356, 2017. URL http://www.aclweb.org/anthology/W17-5039.

[64] Frolin S Ocariza, Jr, Karthik Pattabiraman, and Ali Mesbah. Vejovis: Suggesting fixes for javascript faults. In *Proceedings of the 36th International Conference on Software Engineering*, pages 837–847, 2014.

[65] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. Pytorch, 2017.

[66] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*, pages 547–553. Springer, 2015.

[67] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. Does genetic programming work well on automated program repair? In *2013 International Conference on Computational and Information Sciences*, pages 1875–1878. IEEE, 2013.

[68] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36. ACM, 2015.

[69] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. Elixir: effective object oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 648–659. IEEE Press, 2017.

[70] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. Bugs. jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 10–13, 2018.

[71] Seemanta Saha, Ripon K Saha, and Mukul R Prasad. Harnessing evolution for multi-hunk program repair. In *Proceedings of the 41st International Conference on Software Engineering*, pages 13–24. IEEE Press, 2019.

[72] Keisuke Sakaguchi, Matt Post, and Benjamin Van Durme. Grammatical Error Correction with Neural Reinforcement Learning. 2017. URL http://arxiv.org/abs/1707.00299.

[73] Eddie A Santos, Joshua C Campbell, Abram Hindle, and José Nelson Amaral. Finding and correcting syntax errors using recurrent neural networks. *PeerJ PrePrints*, 2017.

[74] Allen Schmaltz, Yoon Kim, Alexander M. Rush, and Stuart M. Shieber. Adapting Sequence Models for Sentence Correction. 2017. URL http://arxiv.org/abs/1707.09067.

[75] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 99–108. IEEE, 2015.

[76] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 180–182. IEEE Press, 2017.

[77] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM*, pages 832–837, 2018.

[78] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[79] Jinyong Wang and Ce Zhang. Software reliability prediction using a deep learning model based on the rnn encoder–decoder. *Reliability Engineering & System Safety*, 2017.

[80] Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic neural program embedding for program repair. *arXiv preprint arXiv:1711.07163*, 2017.

[81] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 297–308. IEEE, 2016.

[82] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1–11. ACM, 2018.

[83] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 334–345. IEEE, 2015.

[84] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98. ACM, 2016.

[85] Tong Xiao, Tian Xia, Yi Yang, Chang Huang, and Xiaogang Wang. Learning from massive noisy labeled data for image classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2691–2699, 2015.

[86] Qi Xin and Steven P Reiss. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 660–670. IEEE Press, 2017.

[87] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering*, pages 416–426. IEEE Press, 2017.

[88] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2017.

[89] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 831–841. ACM, 2017.

[90] Helen Yannakoudakis, Marek Rei, Øistein E Andersen, and Zheng Yuan. Neural Sequence-Labelling Models for Grammatical Error Correction. *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2795–2806, 2017. doi: 10.18653/v1/D17-1297. URL http://aclweb.org/anthology/D17-1297.

[91] Yuan Yuan and Wolfgang Banzhaf. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering*, 2018.